Opinnäytetyö AMK

Tieto- ja viestintätekniikka

2018

Kristian Koskinen

ANGULAR-PROJEKTIN TESTAAMINEN

OPINNÄYTETYÖ AMK

TURUN AMMATTIKORKEAKOULU

Tieto – ja viestintätekniikka

2018 | 30

Ohjaaja: yliopettaja Mika Luimula, dos.

Kristian Koskinen

ANGULAR-PROJEKTIN TESTAAMINEN

Opinnäytetyön tavoitteena oli tutkia Angular-projektin testaamista ja kehittää valmiina olevaan projektiin käytettäviä testejä sekä luoda pohja näiden testien jatkokehittämiselle. Tarkoituksena oli myös, että tätä työtä voisi käyttää myös pohjana sekä tulevien, että nykyisten projektien testaamiselle. Sekä tavoitteena oli myös tietysti inhimillisten virheiden vähentäminen.

Opinnäytetyön toimeksiantajana toimi NordicEdu Oy ja projektina Angular-ohjelmistokehyksellä asiakkaalle kehitettyyn www-sovellukseen Rainmaker APP:iin. NordicEdu on opetus- ja hyötypelejä valmistava turkulainen yritys, jonka suurin osa projekteista on asiakasprojekteja.

Tutkimuksessa haettiin vastauksia seuraaviin kysymyksiin: Miten testejä kannattaa ruveta kehittämään ja kuinka kannattavaa se on? Kun kyseessä on jo valmiina oleva projekti, täytyykö testien kehittämistä miettiä eri tavalla kuin testivetoisessa kehityksessä? Onko Angularissa hyvät testauskirjastot ja mikä olisi paras tapa käyttää näitä kirjastoja?

Opinnäytetyössä kehitettiin testejä tutkimuksen pohjalta ja luotiin pohja näiden testien jatkokehittämiselle Rainmaker APP:ssä ja tulevissa projekteissa. Opinnäytetyössä myös tarkasteltiin muita samankaltaisia ohjelmistokehyksiä kuin Angular ja testaamisen kannattavuudesta yleensä. Opinnäytetyön tuloksena syntyi käytettäviä testejä ja pohja näiden testien jatkokehittämiselle.

ASIASANAT:

Angular, yksikkötestaaminen, testaaminen

THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information– and Communication Technology

2018 | 30

Supervisor: Principal Lecturer Mika Luimula, Adj. Prof.

Kristian Koskinen

TESTING OF THE ANGULAR FRAMEWORK

NordicEdu Ltd served as the commissioner of this thesis and the project was called Rainmaker APP, developed with the Angular framework. Rainmaker APP is gamified solution for employee tracking. NordicEdu is a Turku-based company which makes serious games and the majority of their projects are customer projects.

The objective of the thesis was to study the testing of the Angular framework, to develop tests used for Rainmaker APP, and to provide the basis for the further development of these tests in current and future projects as well as to reduce human mistakes.

The theoretical part of the thesis discusses the advantages and disadvantages of developing tests with the Angular framework. It also discusses the difference between test-driven-development and already ready project when it comes to developing tests. The thesis also studies Angular test libraries and their usage.

In the thesis, tests were developed based on the literature review and a basis for the further development of these tests was provided for the Rainmaker APP and future projects. The thesis also examined other software frames similar to Angular as well as the profitability of testing in general. It was found that tests are more profitable when they are developed before the tested code. The result of the thesis was the creations of tests to be used and the basis for the further development of these tests.

KEYWORDS:

Angular, unit testing, testing

SISÄLTÖ

KÄYTETYT LYHENTEET

API	Application programming interface, ohjelmointirajapinta
BDD	Behavior-driven-development, yksi testivetoisenkehityksen alalajeista
CLI	Command-line-interface, komentokehotteelta käytettävä rajapinta
IDE	Integrated development environment, Ohjelmointiympäristö (esimerkiksi Microsoft Visual Studio)
JS	JavaScript, www-ympäristöissä käytettävä kieli
REST	hypertekstin siirtoprotokollaan perustuva arkkitehtuurimalli ohjelmointirajapintojen toteuttamiseen
RxJS	RxJS on ReactiveX-kirjasto JavaScriptille
TS	TypeScript, JavaScriptin ylijoukko

1 JOHDANTO

Ennen kaikki internetsivut tekivät pyynnön palvelimelle, joka sitten palautti koko internetsivun näytettäväksi. Palvelimet tekivät kaiken työn dynaamisten sivujen generoinnista. Kaikki tämä muuttui, kun Microsoft julkaisi XMLHttpRequestin. Aluksi tätä käytettiin muuttamaan sivun sisällä olevien yksittäisten elementtien sisältöä. Nykyään www-sovellusten kehittämisessä käytetään yhä kasvavissa määrin logiikan suorittamista asiakkaan puolelle. Palvelimeksi riittää nykyään kevyt ohjelmistokehys REST arkkitehtuurilla, kun asiakaspuoli hoitaa kaiken näkymän logiikan. Tämän takia myös asiakaspuolen testaaminen on yhä tärkeämpää. (Graetz 2018.)

Testaaminen on kovassa suosiossa jatkuvasti kasvavassa ohjelmistokehityksen alalla. Monet ohjelmistokehykset tukevat näiden testien kirjoittamista sekä suorittamista. Yksi näistä on suosittu JS-ohjelmistokehys (JavaScript-ohjelmistokehys) Googlelta, Angular. Tavoitteena oli tutkia Angular-projektin testaamista ja kehittää Angularilla tehtyyn NordicEdu Oy:n projektiin Rainmaker APP:iin käytettäviä testejä sekä luoda pohja näiden testien jatkokehittämiselle. NordicEdussa Angular otettiin käyttöön ennen Rainmaker APP:iä, mutta itse testikirjastojen käyttäminen on jäänyt pois. Tarkoituksena oli myös, että tätä työtä voisi käyttää myös pohjana sekä tulevien, että nykyisten projektien testaamiselle. Sekä tavoitteena oli myös tietysti inhimillisten virheiden vähentäminen.

Miten testejä kannattaa ruveta kehittämään ja kuinka kannattavaa se on? Kun kyseessä on jo valmiina oleva projekti, täytyykö testien kehittämistä miettiä eri tavalla kuin testivetoisessa kehityksessä? Entä tarjoaako Angular hyvät testauskirjastot ja miten niitä kuuluu käyttää? Muita opinnäytetöitä ei tästä aiheesta juurikaan ole tehty, tosin Angularin vanhemman version testaamisesta on.

Aluksi opinnäytetyössä käydään läpi JS-ohjelmistokehyksiä ja etenkin Angularia sekä siihen liittyviä osia. Tämän jälkeen käydään läpi Angularin mukana tulevia testikirjastoja ja niiden käyttöä sekä yleisesti testaamisen kannattavuudesta. Lopuksi käydään läpi itse Rainmaker APP:in testausprosessi.

2 JS-OHJELMISTOKEHYKSET

2.1 JS-ohjelmistokehys

JS-ohjelmistokehyksen valitseminen alkaa nykyään olla melko hankalaa kehyksien suuren lukumäärän takia. Suosituimmat Angular ja React jatkavat suosionsa kasvua, mutta myös uudemmat ja pienemmät ohjelmistokehykset ovat alkaneet saada huomiota. Tässä työssä käsitellään front-end yksittäissivu-www-sovellusten kehitysalustoja. Ero yksittäis-sivu ja moni-sivun välillä on se, että yksittäis-sivu-sovelluksessa koko sivua ei ladata uudestaan, kun linkkiä tai nappulaa painetaan, vain ainoastaan muuttuvat komponentit, koska kaikki tarvittava ladataan kerralla. (Neuhaus 2017.)

2.2 Ohjelmistokehysten hyödyt

Ohjelmistokehykset ovat vain työkaluja, jolla työntekijä tekee työnsä paremmin. Hyvä ohjelmoija tietää, miten tehdä asiat hyvin ja nopeasti. Jos koodia ei ole pakko kirjoittaa, niin miksi silloin pitäisi? Ohjelmistokehykset antavat tällaisia valmiita paketteja, jotka vähentävät työntekijän työtaakkaa ja antavat hänen keskittyä olennaiseen. Varsinkin aloittelevalle kehittäjälle ohjelmistokehykset ovat pelastus, kun lähes varmasti joku on jo tehnyt tarvitun asian ja paremmin kuin olisi tehnyt itse. (Andras 2017.)

NordicEdu päätyi Angulariin, koska uskoi sen olevan paras vaihtoehto silloiseen projektiin ja näki Angularin käyttöönoton olevan hyödyllistä myös tulevien projektien kannalta.

2.3 Angular-ohjelmistokehys

Angular on TS-pohjainen (TypeScript) front-end-www-sovellusten ohjelmistokehys. Angularia voisi kuvailla, että se olisi HTML ja JS niin kuin ne olisi pitänyt tehdä sovelluksia varten. Suurien www-sovellusten kehittäminen ilman Angularia olisi todella vaikeata sekä vaatisi paljon resursseja ylläpitää. (Angular 2018.)

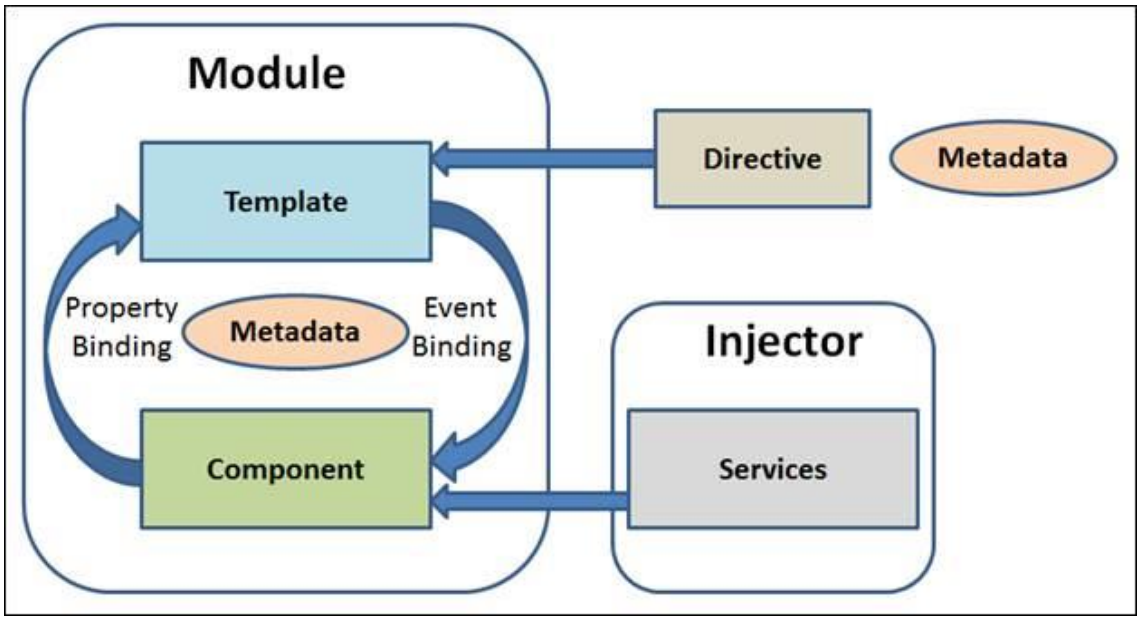
Angular CLI:tä (command-line-interface) käyttäessä, tulee projektia luodessa myös automattisesti testikirjastot ja -tiedostot testaamista varten. Jasmine on JS-pohjainen

kirjasto testien kirjoittamiselle. Samalla tulee myös työkalu näiden testien suorittamiseen selaimessa.

Angularin edeltäjä AngularJS julkaistiin jo vuonna 2010, mutta se ei saanut yhtä suurta suosiota heikon laadun ja käytännöllisyyden puutteen takia. Google päätti tehdä kaiken uudestaan ja teki AngularJS pohjalta Angularin, joka julkaistiin syyskuussa 2016. Edelleen näkee alan artikkeleissa puhuttavat väärin Angularista AngularJS:nä, joka voi aiheuttaa hämmennystä, varsinkin asiasta tietämättömälle.

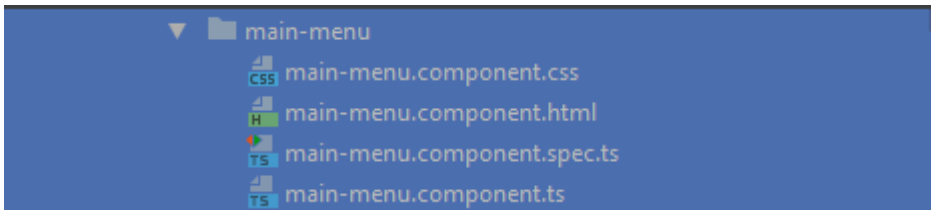
AngularJS:ssä scope on tavallinen JS-objekti, johon voidaan kontrollerilla asettaa aloitustila tai lisätä toiminnallisuutta. Angularissa ei ole käsitettä scopeista ja kontrollereista niin kuin AngularJS, sen sijaan se käyttää komponenttien hierarkiaa. Komponenttien hierarkialla tarkoitetaan, että toisen komponentin sisällä voidaan käyttää toista komponenttia. Hierarkia suunniteltaessa on kuitenkin muistettava välttämään kehäriippuvuutta. Tämä tarkoittaa esimerkiksi, että komponentti A on riippuvainen komponentti B:stä, joka on riippuvainen komponentti C:stä, joka taas on riippuvainen A:sta, ja näin syntyy loputon kehä. (AngularJS 2018)

Angular-sovellus koostuu 8 eri palikasta. Kuva 1 näyttää Angularin arkkitehtuurin. Niistä tärkeimmät ovat moduulit (Module), komponentit (Component) ja palvelut (Service). Moduuleita voi ajatella säiliönä kaikille komponenteille ja palveluille. Palvelu on funktio tai objekti, jota käytetään datan jakamiseen ja käsittelyyn koko applikaation sisällä. Komponentit sisältävät näkymän logiikan ja itse näkymän. (Trivedi 2016)



Kuva 1 Angular sovelluksen arkkitehtuuri (Trivedi 2016).

Angularissa komponenttien hierarkia on tärkeimmässä osassa sen arkkitehtuuriselle ominaisuudelle. Kuva 2 näkyy automaattisesti luodun komponentin tiedostorakenne. Ylimpänä tiedostoista on CSS-tiedosto. CSS on kieli, joka kuvailee HTML-tiedoston tyylejä ja kuinka HTML-elementit tulisi sijoittaa näkymässä. Toisena on komponentin HTML tiedosto. HTML on avoimesti standardoitu kuvauskieli, jolla kuvataan hypertekstiä. Tunnetuksi HTML on tullut siitä, että sillä kirjoitetaan internetsivut. Sen jälkeen tulee Jasminen testaustiedosto. Sinne kirjoitetaan kaikki testausta varten logiikka ja itse suoritettavat testausfunktiot. Alimpana on komponentin TS tiedosto, johon kirjoitetaan kaikki komponentin logiikka ja muut funktiot.



Kuva 2: Automaattisesti luodun komponentin tiedostorakenne.

2.4 React-ohjelmistokehys

React on JS kirjasto käyttäjärajapintojen kehittämiseen. Sillä on hyvät tuet natiiveihin sovelluksiin, minkä takia se onkin suosittu mobiilisovellusten kehittäjien kanssa.

Maaliskuussa 2013 julkaistu React on Facebookin kehittämä ja ylläpitämä. Facebook käyttää React komponentteja monilla sivuilla. Se on käytössä Facebookilla paljon enemmän kuin Angular on käytössä Googlella. Reactia käyttää mm. Uber, Netflix, Twitter, Reddit, Paypal, Imgur ja monet muut suuret palvelut.

2.5 Vue-ohjelmistokehys

Vue on yksi nopeimmin kasvavista JS kehitysalustoista. Vueta kuvaillaan intuitiiviseksi, nopeaksi ja helposti kääntyväksi malli-näkymä-mallinäkymäksi interaktiivisten rajapintojen kehittämiseen. Se julkaistiin helmikuussa 2014 Googlen ex-työntekijän Evan You toimesta. Vue on ollut suuri menestys, ottaen huomioon, että se oli pitkään yhden henkilön ylläpitämä ilman suuren yhtiön tukea. Nykyään Youlla on noin tusina kehittäjää apunaan. Vueta käyttää mm. Alubab, Nintendo, GitLab ja monet muut pienemmät projektit.

2.6 Suurimmat erot

Kaikki edellä mainitut ohjelmistokehykset ovat komponenttipohjaisia. Komponentti saa sisäänsä dataa ja oman sisäisen toiminnan jälkeen palauttaa generoidun käyttäjänäkymän pohjan. Komponentit pitäisivät olla helposti uudelleen käytettävissä nettisivuilla ja muissa komponenteissa.

React käyttää EcmaScript6, kun Vue käyttää joko EcmaScript5 tai EcmaScript6. Angular sen sijaan käyttää TS. Tämä mahdollistaa dekoraattorit ja staattiset tyyppimäärittelyt.

Angular on enemmänkin ohjelmistokehys kuin kirjasto, koska se tarjoaa vahvoja mielipiteistä siitä, miten sovellus tulisi rakentaa. Angular on valmis pakkaus, ei tarvitse analysoida kirjastoja tai muuta sellaista, vaan voi suoraan aloittaa työn tekemisen.

Kun data komponentissa muuttuu, joutuu se generoimaan uudestaan koko komponentin. Vue käyttää tähän datan kiinnittämiseen ja tilanhallintaan Vuexia, mutta siinä voi käyttää myös Reduxia. React ja Angular käyttävät molemmat Reduxia, mutta suurin ero niiden välillä on yksisuuntainen vastaan kaksisuuntainen datan kiinnittäminen. Tarkoittaen että Angularia käyttäessä tila voi muuttua myös vaikkapa käyttäjän inputista.

2.7 TypeScript-kieli

TS on avoimenlähdekoodin ohjelmointi kieli, jota ylläpitää Microsoft. TS on JS kielen ylijoukko ja se on suunniteltu isojen sovellusten kehittämiseen ja se kääntyy JS:lle. Tämä tarkoittaa myös sitä, että kaikki olemassa olevat JS ohjelmat ovat myös päteviä TS ohjelmia. Kahden vuoden sisäisen kehittämisen jälkeen, lokakuussa 2012 julkaistu TS, sai ylistystä alan tekijöiltä, mutta risuja huonon IDE (Integrated development environment, Ohjelmointiympäristö) tuen takia, jonka takia sitä ei ollut saatavilla kuin Windowsille ja sen Visual Studiolle. Nykyään moni IDE tukee TS. (Microsoft 2018)

Suurimmat asiat mitä TS tuo ovat tyyppimäärittelyt, deklaraatiosta ja luokat. Kuvissa 3-5 koodiesimerkit jokaisesta. Kuva 3 on esimerkki tyyppimäärittelyistä. Siinä funktio ottaa parametreiksi kaksi numeroa, vähentää ne toisistaan ja palauttaa numeron. Kuva 4 on esimerkki deklaraatiosta. Siinä luodaan rajapinta tyypille LaskuToimitukset. Kuva 5 on esimerkki luokasta. Siinä annetaan henkilölle kentät nimi, ikä ja pituus ja konstruoija Henkilön luomista varten sekä metodi henkilön tietojen hakuun.

```
// Funktio ottaa sisäänsä kaksi numeroa ja palauttaa numeron
vahenna(v: number, o: number): number {
  return v - o;
}
```

Kuva 3 Esimerkki tyyppimäärittelyistä.

```
// Declarationit toimivat kuin rajapinnat
declare interface LaskuToimitukset {

  lisaa(y: number, o: number): number;
  vahenna(y: number, o: number): number;
  kerro(y: number, o: number): number;
  jaa(y: number, o: number): number;

}
```

Kuva 4 Esimerkki deklaraatiosta.

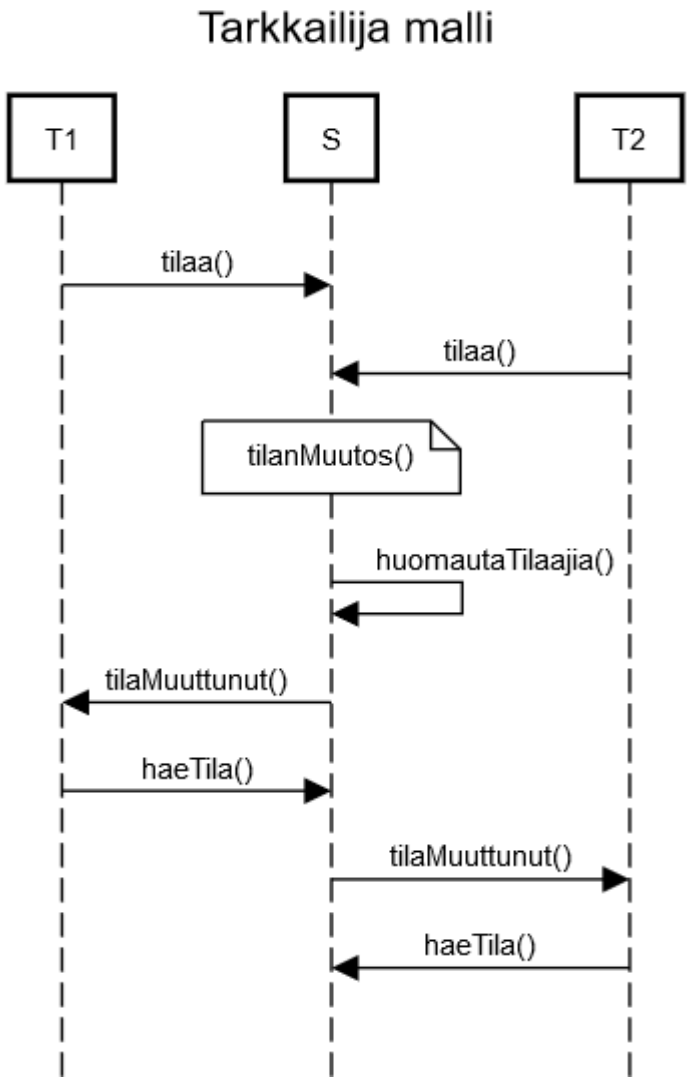
```
class Henkilo {
  private nimi: string;
  private ika: number;
  private pituus: number;

  constructor(nimi: string, ika: number, pituus: number) {
    this.nimi = nimi;
    this.ika = ika;
    this.pituus = pituus;
  }

  haeTiedot(): string {
    return `${this.nimi} (${this.ika}) (${this.pituus})`;
  }
}
```

Kuva 5 Esimerkki luokasta.

2.8 ReactiveX-kirjasto JavaScriptille

RxJS (ReactiveX-kirjasto JavaScriptille) on Microsoftin ja yhteisön ylläpitämä kirjasto async-datastreamille ja observer-mallille, perustuen Reduxiin. Tämä mahdollistaa asynkroonisen käytön sovellukselle (auttaa kun lähes aina kyseessä on http-kutsujen odottaminen.) Myös testikirjastojen mukana tulee omat testaus metodit RxJS:ä varten. (learnRxJS 2018)

Tarkkailija malli perustuu olioiden väliseen keskusteluun käytettävässä tarkkailijassa, jotka muuttavat tiedon niissä olioissa, jotka ovat siitä riippuvaisia. Kuva 6 on havainnollistettu tarkkailija mallia. Olkoon subjekti *S* sekä tarkkailijat *T1* ja *T2*. Tarkkailijat tilaavat subjektin tilan. Tällöin, kun S:n tilaa muutetaan, lähetetään T1 ja T2 ilmoitus muuttuneesta tilasta, jolloin T1 ja T2 hakevat S:n tilan uudelleen.



Kuva 6. Sekvenssi-diagrammi tarkkailija mallin esimerkistä.

Tilaajamalli on hyödyllinen, kun moni on riippuvainen yhdestä tilasta. RxJS helpottaa tämän käyttämistä. Alla koodiesimerkki Angularista käyttäen TS:ä ja RxJS. Esimerkissä luodaan luokat S ja T1. S:llä on julkinen kenttä nimeltään Tila, joka on tyypiltään subjekti, ja funktio tilanMuutos, jota kutsumalla voi muuttaa Tilan arvoa ja huomauttaa tilaajia. T1:llä on yksityinen kenttä nimeltään summa ja T1:n konstruoijassa annetaan parametrina luokka S, niin kuin Kuva 7. Kun Tila muuttuu, lisätään se summaan (Kuva 8).

```
class S {
  public Tila: BehaviorSubject<number> = new BehaviorSubject<number>( _value: 1);

  tilanMuutos(i: number) {
    this.Tila.next(i);
  }
}


class T1 {
  private summa: number;

  constructor(private s: S) {
    s.Tila.subscribe( next: tila => this.summa += tila);
  }
}
```

Kuva 7 Esimerkki tilaaja-mallista.

```
// Alussa summa = 1
this.tilanMuutos( i: 2);     // Summa on nyt 3
this.tilanMuutos( i: -4);    // Summa on nyt -1
this.tilanMuutos( i: 0);     // Summa on edelleen -1
```

Kuva 8 Tilaaja-mallin eteneminen.

3 TESTIKIRJASTOT JA TYÖKALUT

3.1 JS-testikirjastot

JS on niin suosittu kieli, että ohjelmistokehyksiä tulee jatkuvast uusia, sama pätee myös testikehyksiin ja -kirjastoihin. Käytetyimpiä ovat Jasmine, Mocha ja Jest. Tässäkin työssä käytetty Jasmine on yksi suosituimmista. Jasmine tukee assertioita, spy-funktioita ja mock-metodeja. Sen mukana tulee siis kaikki, mitä tarvitaan yksikkötestien kirjoittamiseen (Ben 2017). Syy Jasminen käyttöön on se, että Google suosittelee Jasminea ja siksi se tuleekin Anglular-CLI:tä käyttäessä Angular-projektin mukaan automaattisesti.

Jest on Facebookin kehittämä, käyttämä ja suosittelema testauskirjasto. Jestiä käytetään lähinnä React-sovellusten testaamiseen, mutta sen voi helposti integroida myös muihin ohjelmistokehyksiin. Jest on myös todella suorituskykyinen sen rinnakkaisen testaamisen ansiosta. Mocha sen sijaan on kaikista suosituin. Se on joustava kirjasto tarjoten vain perusominaisuudet. Muut kuten spy-funktiot voidaan lisätä erikseen, jos kehittäjä sitä haluaa. Mocha eroaa eniten muista tarjoten globaalin testirakenteen, tarkoittaen ettei tarvitse erikseen jokaiseen tiedostoon lisätä tarvitsemia liitännäisiä. (Ben 2017)

3.2 Jasmine-kirjasto

Jasmine on testausalusta JS:lle, mikä tukee sovelluskehitys tyyliä nimeltään behavior-driven-development. Se on yksi testivetoisen kehityksen tyyleistä. Jasmine ja BDD yleensäkin yrittävät kuvailla testejä ihmiselle luettavassa muodossa, jotta myös ei-niin tekniset ihmiset voisivat myös lukea niitä. Vaikka olisikin tekninen, tekee se silti lukemisesta helpompaa. (Hussain 2017)

Jasminella kirjoitetaan yksikkötestejä. Yksikkötestit ovat testejä jotka kohdistuvat vain tiettyyn osaan isommasta kokonaisuudesta, yleensä vain yhteen funktioon. Näin päästään keskittymään vain tiettyyn osaan koodia ja virheiden etsiminenkin on helpompaa, kun tietää suoraan mistä kohtaan etsiä. (SWTF 2018)

Varsinaista syntaksia ei ole describe- ja it-funktioiden merkkijonoille, mutta kehotus olisi, että funktioissa kerrotaan, mitä testataan yleisellä tasolla (esimerkiksi testattavan komponentin nimi) ja it -funktioissa minkä pitäisi olla mitäkin (esimerkiksi funktio X pitäisi palauttaa Y). Kuva 9 on esimerkki yksinkertaisen funktion testistä.

```
function helloWorld() {
  return "Hello world!";
}

describe( description: "Hello world tests", specDefinitions: () => {
  it( expectation: "should say Hello world!", assertion: () => {
    expect(helloWorld()).toEqual( expected: "Hello world!");
  });
});
```

Kuva 9 Esimerkki yksinkertaisesta testistä.

Testejä kirjottaessa aikaa voi helposti säästää joko ohittamalla tai keskittymällä tiettyihin testeihin. Tämä onnistuu joko laittamalla ”x”, jos haluaa ohittaa, tai laittamalla ”f”, jos haluaa keskittyä vain siihen testiin. Näitä on turha sekoittaa, sillä vain keskittymiset huomioidaan, jos niitä on. Kuva 10 esimerkki keskittymisestä ja Kuva 11 esimerkki ohittamisesta.

```
function add(left, right) {
  return left + right;
}

describe( description: "Add unit tests", specDefinitions: () => {

  // Vain tämä suoritetaan
  fit( expectation: "should return 5 when 2 + 3", assertion: () => {
    expect(add( left: 2, right: 3)).toEqual( expected: 5);
  });

  it( expectation: "should return 0 when 0 + 0", assertion: () => {
    expect(add( left: 0, right: 0)).toEqual( expected: 0);
  });

  it( expectation: "should return -1 when 2 + (-3)", assertion: () => {
    expect(add( left: 2, right: -3)).toEqual( expected: -1);
  });
});
```

Kuva 10 Esimerkki keskittymisestä.

```
function add(left, right) {
  return left + right;
}

describe( description: "Add unit tests",  specDefinitions: () => {

  // Tätä ei suoriteta
  xit( expectation: "should return 5 when 2 + 3",  assertion: () => {
    expect(add( left: 2,  right: 3)).toEqual( expected: 5);
  });

  it( expectation: "should return 0 when 0 + 0",  assertion: () => {
    expect(add( left: 0,  right: 0)).toEqual( expected: 0);
  });

  it( expectation: "should return -1 when 2 + (-3)",  assertion: () => {
    expect(add( left: 2,  right: -3)).toEqual( expected: -1);
  });
});
```

Kuva 11 Esimerkki ohittamisesta.

Yksikkötestaamissa on tärkeää eristää testattava komponentti. Kaikki ulkopuolinen logiikka pitää jäljitellä. Jasminella on tähän käyttöön tarkoitetut spy-funktiot. Spy-funktiot korvaavat oikean funktion ja niille pystyy määrittelemään paluuarvon. Kuva 12 on esimerkki spy-funktion käytöstä. Siinä jäljitellään add-funktiota ja muunnetaan sen palautus arvoa oman mielen mukaan.

```
function add(left, right) {
  return left + right;
}

describe( description: "spy demo",  specDefinitions: () => {
  it( expectation: "should fake the answer",  assertion: () => {
    expect(add( left: 1,  right: 2)).toBe( expected: 3);
    spyOn(this,  method: "add").and.returnValue( val: 4);
    expect(add( left: 1,  right: 2)).toBe( expected: 4);
  });
});
```

Kuva 12 Esimerkki spy-funktion käytöstä.

3.3 Protractor-kirjasto

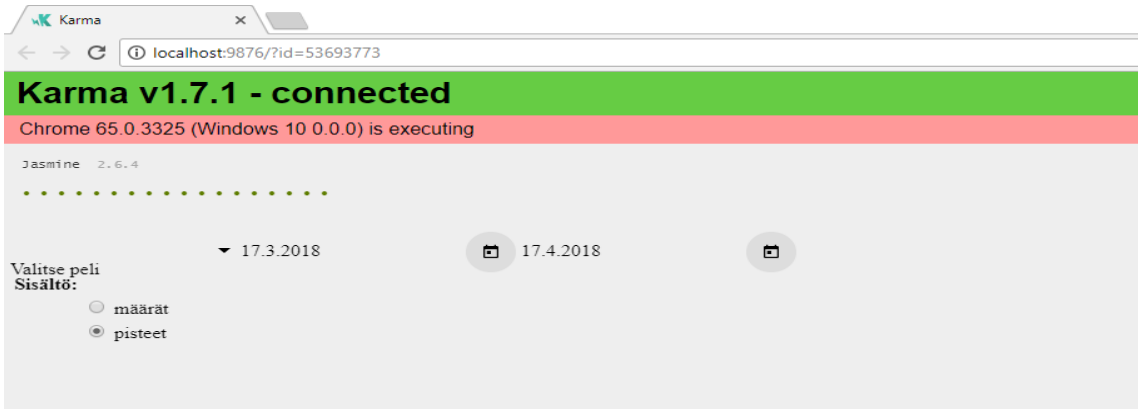
Protractor on käyttöliittymien testauskehys Angularille. Protractor suorittaa testejä oikeassa selaimessa ja mahdollistaa sellaisen käytön kuin oikea käyttäjä sitä käyttäisi. Protractorilla voi muun muassa hakea sivulta kaikki nappulat ja painaa niitä, tai kaikki teksti-inputit ja kirjoittaa niihin. Kuva 13 koodiesimerkki testistä joka testaa sivua jossa voi laskea kahden luvun summan. (Protractor 2018)

```
describe( description: "Protractor demo", specDefinitions: () => {
  it( expectation: "should give correct result", assertion: () => {
    browser.get( destination: "sivun/osoite/tähän");
    element(by.id("n1")).sendKeys(1);
    element(by.id("n2")).sendKeys(3);
    element(by.id("laske-nappula")).click();
    expect(element(by.id("tulos")).getText()).toEqual( expected: "4");
  });
});
```

Kuva 13 Protractor esimerkki

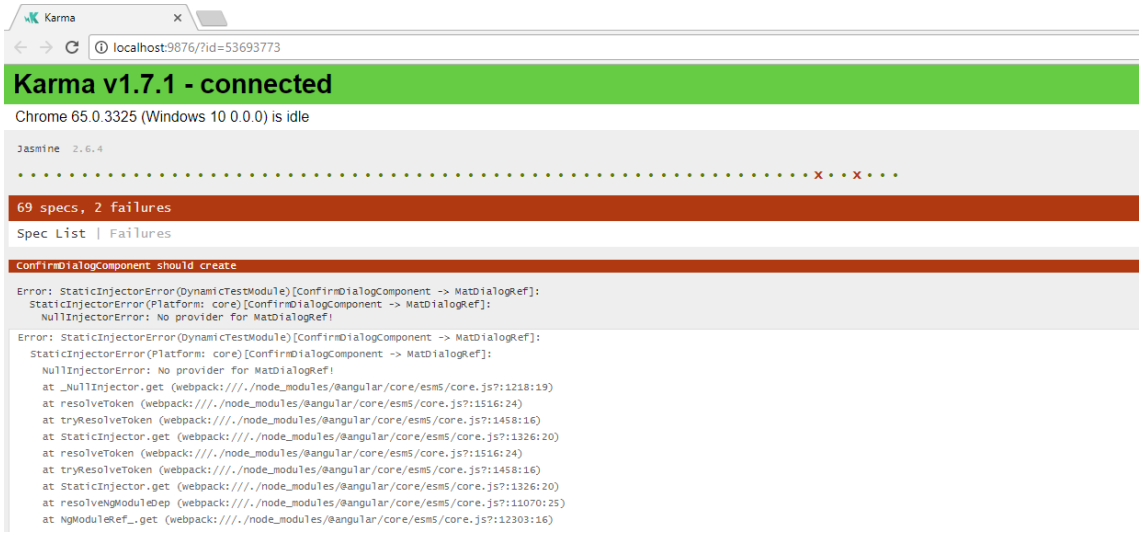
3.4 Karma-työkalu

Kun koodia muutetaan, on epämiellyttävää suorittaa Jasmine-testejä manuaalisesti päivittämällä jatkuvasti selaimen sivua. Karma on väline joka mahdollistaa selaimen käynnistämisen ja näiden testien suorittamisen automaattisesti aina kun koodia muutetaan. Kuva 14 näkyy Karma suorittamassa näitä testejä (Karma 2018).



Kuva 14 Karma suorittamassa testejä.

Itse testien tulokset näkyvät sekä komentokehotteessa että selaimessa. Selaimessa on helpompi lukea tuloksia sen paremman visuaalisuuden takia. Kuva 15 näkyy yhden testikierroksen tulos. Yläosassa näkyvät eriväriset merkit merkitsevät suoritettuja testejä ja väri kertoo mitkä niistä onnistuivat ja mitkä epäonnistuivat. Sen jälkeen lukee epäonnistuneiden testien virheviesti, mistä voi päätellä mikä on mennyt pieleen.



Kuva 15 Karman testauksen tulos.

3.5 Mitä kannattaa testata ja mitä ei?

Melkein mitä vain voi testata, mutta aina ei ole kannattavaa käyttää aikaa kaiken testaamiseen. Kaikki logiikkaa vaativat ovat lähes pakollisia testata, varmistaakseen ettei niihin tule ohjelman toiminnan kannalta haittaavia virheitä. Testeissä kannattaa ottaa huomioon mahdolliset ääripäät sisääntulossa ja palautusarvossa. (Kolodiy 2018)

Testata kannattaa vain testattavan komponentin toimintaa, ei siihen liittyvien komponenttien. Muiden komponenttien testaus on sitten niiden komponenttien sisällä. Myöskään mitään api-kutsuja ulospäin ei kannata testata, kun tarkoituksena on eristää mahdollisimman hyvin komponentti. Kuitenkin kannattaa testata api-kutsun vastauksen käsittely.

4 TESTIEN KIRJOITTAMISEN KANNATTAVUUS

Testaamisen kannattavuudesta on tehty tutkimuksia, mutta kiistatonta näyttöä yksikkötestauksen kannattavuudesta ei juurikaan ole. Ammattikirjallisuudessa testaamista pidetään tehokkaana ja hyödyllisenä. Kannattavuus on kuitenkin vaikea määritellä. Tätä on vaikea testata, koska tutkimuksissa yleensä keskitytään yhteen projektiin. Ja samaa projektia ei voi sama henkilö tehdä kahdesti, mikä tekee objektiivisen vertailun vaikeaksi. (Haltiapuu 2016)

Testien kehittämisen haitat keskittyvät ylimäärisen työn määrään. Testien kirjoittaminen ja ylläpitäminen vievät aikaa ja resursseja. Yleensä sama henkilö kirjoittaa itse testin sekä testattavan koodin, ja näin syntyy ns. sokeita kohtia. Suuren yksikkötestaus määrän takia voi tulla väärä usko toimivuudesta, minkä takia sivuutetaan muita olennaisia testaamisia. (Dalke 2009)

Testien kehittämisen hyödyt keskittyvät testeistä saatuun luottamukseen sovelluksen toiminnasta. Yksikkötestien kirjoittaminen mahdollistaa isoja muutoksia koodiin nopeasti, koska testien suorittaminen varmistaa toiminnallisuuden. Testivetoinen kehitys auttaa ymmärtämään mihin on hyvä lopettaa, koska testit antavat luottamusta toiminnasta jolloin tietää, että voi lopettaa koodin hiomisen ja siirtyä eteenpäin. Yksikkötestit auttavat ymmärtämään, mitä ylipäätään juuri siltä koodinpätkältä vaaditaan, kun enne koodin kirjoittamista määritellään ehdot ja mitä koodin oletetaan palauttavan. Useimmat IDE:t tukevat yksikkötestien suorittamista ja antavat siitä visuaalisen palautteen (Kuva 15), pelkkää vihreätä kertoo, että kaikki on hyvin, punainen kertoo mistä aloittaa työskentely. Yksikkötestit tekevät koodista modulaarisemman. Samaa koodia ja samoja testejä voi käyttää myös muissa projekteissa helposti. (Reefnet 2014)

Testien kirjoittaminen auttaa myös yleisen koodin kirjoittamistaidon kehitystä. Testien kirjoittamisen jälkeen alkaa nähdä, jos jossain kohdassa on virhe sen enempää sitä testaamatta. Tällaiset toistuvat pienet virheen saadaan helposti pois testaamisen avulla.

5 RAINMAKER APP TESTAUS

5.1 Suunnitelma

Aluksi kävin materiaalia läpi internetistä, jotta saisin käsityksen, mitä tässä oikein ollaan tekemässä. Angularin omassa dokumentaatiossa on hyvät ohjeet testien kirjoittamiselle, muttei sellaiselle, joka niitä ei ole ennen kirjoittanut. Heidän dokumentaatiossa oli kohtia, joissa luki ”Tulossa pian”, mikä kertoo, että ollaan aika uutta asiaa tutkimassa. Onneksi internetistä löytyi artikkeleita, jotka kävivät testien kirjoittamisen kohta kohdalta läpi, selittäen mitä missäkin kohdassa tapahtuu ja miksi niin tehdään.

Tämän jälkeen tein suunnitelman siitä, mistä aloittaisin kirjoittamisen, miten etenisin ja mihin lopettaisin. Päätin aloittaa siitä, että otan listalta ensimmäisen komponentin ja tutkisin, mitä kaikkea siitä kannattaa testata ja kirjoittaa testit näiden testaamiseksi. Sen jälkeen tulisin etenemään seuraavaan komponenttiin ja niin edelleen, kunnes komponentin loppuisivat tai uskon, ettei testaamisella saavutettaisi enempään lisäarvoa.

5.2 Testien kirjoittaminen

Itse testien kirjoittamisen aloitin sillä, että sain automattisesti luodun testitiedoston menemään läpi tarkastuksen. Näissä testeissä testataan vain yhtä asiaa: voiko komponentin luoda. Tämä tarkoittaa, että testitiedostoon pitää lisätä kaikki komponentin riippuvuudet. Tämä osoittautui yllättävän vaikeaksi, koska kaikkea ei voinut lisätä sellaisenaan, vaan piti käyttää kyseisen komponentin tai kirjaston testiversiota.

Testit alkoivat nopeasti muistuttamaan toisiaan, niinpä käyn tässä läpi vain muutaman tärkeimmän kohdan. Totta kai jokainen komponentti on erilainen, mutta perusidea testien takana on sama kaikissa.

5.2.1 Piippujen testaaminen

Piiput ovat eräänlaisia Angularin komponentteja, joilla ei ole näkymää. Piiput ottavat sisäänsä dataa ja muuttavat sen halutuksi ulostuloksi. Piippuja kannattaa käyttää, kun halutaan välttää funktioiden käyttö HTML-tiedoston sisällä, jolloin funktiota kutsuttaisiin

aina kun käyttäjä tekee jotain, näin hidastavan sovellusta. Niiden testaaminen on yksinkertaisin asia Angularin testaamisesta. Kuva 16 on yksi Rainmaker APP:ssä käytetty piippu. Siinä muutetaan saatu data prosenttiluvuksi. Data on muotoa taulukko, jonka sisällä on objekteja, joiden kentissä on arvoja ja taulukon viimeisenä on summa näistä arvoista.

```
export class CellToPercentagePipe implements PipeTransform {
  transform(row?: any, args?: any[]): string {
    if (!row || !args || args.length < 2 ||
        !row[args[0]] || !row[args[0]][args[1]] ||
        !row["total"] || !row["total"][args[1]]) {
      return "(0.0%)";
    }
    return "(" + (row[args[0]][args[1]] * 100 / row["total"][args[1]]).toFixed( fractionDigits: 1) + "%)";
  }
}
```

Kuva 16 Piippu, joka muuttaa taulukon solun datan prosenteiksi

Piipun testien kirjoittaminen lähti siitä, että piipulle annetaan oikean tyyppistä dataa ja tarkastellaan, että piippu palauttaa oikein. Kuva 17 on tästä kehitetty testi. Sen jälkeen piti tietenkin testata virhetilanteet. Ensin piipulle annetaan tyhjää dataa ja tarkistetaan palautusarvo. Sen jälkeen piipulle annetaan virheellistä dataa ja tarkistetaan palautusarvot jälleen. Kuva 18 on tästä kehitetty testi.

```
it( expectation: "should return correct value", assertion: () => {
  const pipe = new CellToPercentagePipe();
  const row = {
    testi1: {tyyppi1: 1, tyyppi2: 2},
    testi2: {tyyppi1: 3, tyyppi2: 4},
    total: {tyyppi1: 4, tyyppi2: 6}
  };
  expect(pipe.transform(row, args: ["testi1", "tyyppi1"])).toBe( expected: "(25.0%)");
  expect(pipe.transform(row, args: ["testi2", "tyyppi1"])).toBe( expected: "(75.0%)");
  expect(pipe.transform(row, args: ["testi2", "tyyppi2"])).toBe( expected: "(66.7%)");
  expect(pipe.transform(row, args: ["testi1", "tyyppi2"])).toBe( expected: "(33.3%)");
});
```

Kuva 17 Testi, että piippu palauttaa oikean arvon

```
it( expectation: "should return (0.0%) if something is wrong", assertion: () => {
  const pipe = new CellToPercentagePipe();
  expect(pipe.transform()).toBe( expected: "(0.0%)");
  expect(pipe.transform( row: 0)).toBe( expected: "(0.0%)");
  expect(pipe.transform( row: "0")).toBe( expected: "(0.0%)");
  expect(pipe.transform( row: {}, args: [])).toBe( expected: "(0.0%)");
  const row = {
    testi1: {tyyppi2: 2, tyyppi3: 3},
    testi2: {tyyppi1: 3, tyyppi2: 4},
    total: {tyyppi1: 4, tyyppi2: 6}
  };
  expect(pipe.transform(row, args: ["testi1", "tyyppi1"])).toBe( expected: "(0.0%)");
  expect(pipe.transform(row, args: ["testi2", "tyyppi3"])).toBe( expected: "(0.0%)");
  expect(pipe.transform(row, args: ["testi3", "tyyppi2"])).toBe( expected: "(0.0%)");
  expect(pipe.transform(row, args: ["testi1", "tyyppi3"])).toBe( expected: "(0.0%)");
});
```

Kuva 18 Testi, että piippu palauttaa oletusarvon virhetilanteissa

5.2.2 Komponenttien testaaminen

Rainmaker APP:ssä komponentit hakevat datansa palveluilta ja muuntavat sen käyttäjälle haluttuun muotoon. Opinnäytetyössä ei käydä läpi palveluiden testaamista, koska ne vain toimivat välikätenä komponenttien ja palvelimen välillä. Kuva 19 on alkuperäinen funktio. Funktiossa mukana oleva ”this.errorText” käytetään näyttämään käyttäjälle virheviesti. Tätä on vaikea ruveta testaamaan, koska funktio ei anna palautetta, missä virhe tapahtui. Tästä syystä erotin itse virheen etsimisen erikseen sekä virheviestin toiseen. Kuva 20on muokatut funktiot. Koitin tehdä funktioista mahdollisimman puhtaita, jotta ne olisivat helposti myös testattavissa. Itse käyttäjän datan tarkistus palauttaa StatusCoden, jota käytetään sitten virheviestin määrittelyyn.

```
public checkData(user: User): boolean {
  if (!user.email || user.email.length === 0) {
    this.errorText = "Käyttäjätunnus ei voi olla tyhjä";
    return false;
  }
  if (!user.name || user.name.length === 0) {
    this.errorText = "Käyttäjän nimi ei voi olla tyhjä";
    return false;
  }
  return this.checkGoal(user);
}
```

Kuva 19 Alkuperäinen funktio

```
public checkUserDataStatus(user: User): StatusCode {
  if (!user.email || user.email.length === 0) {
    return StatusCode.InvalidEmail;
  }
  if (!user.name || user.name.length === 0) {
    return StatusCode.InvalidName;
  }
  return StatusCode.Success;
}

public errorMsg(code: StatusCode): string {
  switch (code) {
    case StatusCode.InvalidEmail:
      return "Käyttäjännus ei voi olla tyhjä";
    case StatusCode.InvalidName:
      return "Käyttäjän nimi ei voi olla tyhjä";
    default:
      return "Tuntematon virhe";
  }
}

public checkData(user: User): boolean {
  const status = this.checkUserDataStatus(user);
  if (status !== StatusCode.Success) {
    this.errorText = this.errorMsg(status);
    return false;
  }
  this.errorText = "";
  return true;
}
```

Kuva 20 Muokatut funktiot

Kuva 21 on luodut testit. Virheviestin palauttavan funktion testaaminen on helppoa. Pitää vain varmistaa, että vain virhekoodeilla saadaan oikea virheviesti ja muilla palautetaan ”tuntematon virhe”. Datan tarkistamiseen käytetty funktio oli myös helppo, mutta hieman erilainen. Alussa funktiolle annetaan tyhjä Käyttäjä-olio. Tällöin funktion pitää palauttaa virheviesti. Sen jälkeen Käyttäjä-oliolle annetaan oikeat tiedot kenttiin ja varmistetaan, että funktio palauttaa oikein.

```
  it("should return correct errorMsg", () => {
    expect(component.errorMsg(StatusCode.InvalidEmail)).toBe("Käyttäjännus ei voi olla tyhjä");
    expect(component.errorMsg(StatusCode.InvalidName)).toBe("Käyttäjän nimi ei voi olla tyhjä");
    expect(component.errorMsg(StatusCode.Success)).toBe("Tuntematon virhe");
    expect(component.errorMsg(0)).toBe("Tuntematon virhe");
    expect(component.errorMsg(5)).toBe("Tuntematon virhe");
  });

  it("should return correct StatusCode when checking user data", () => {
    const user = new User();
    expect(component.checkUserDataStatus(user)).not.toBe(StatusCode.Success);
    user.email = "testi@test";
    expect(component.checkUserDataStatus(user)).not.toBe(StatusCode.Success);
    user.name = "Testaaja";
    expect(component.checkUserDataStatus(user)).toBe(StatusCode.Success);
    user.name = "";
    expect(component.checkUserDataStatus(user)).toBe(StatusCode.InvalidName);
    user.name = "Testaaja";
    user.email = "";
    expect(component.checkUserDataStatus(user)).toBe(StatusCode.InvalidEmail);
  });
});
```

Kuva 21 Käyttäjän datan tarkistuksen testit

Komponentin näkymää testataan Protractorilla. Useimmat Protractor-testit olivat sellaisia, missä haettiin tekstikentät, syötettiin niihin generoituja sanoja ja painettaisiin nappuloita. Kuva 22 on käyttäjänäkymän yksi Protractor-testeistä. Siinä tutkitaan nappuloiden painamista, kun tekstikentissä ei ole mitään sisältöä. Ensin haetaan kaikki nappulat ja testataan, että oikeat funktiot on kutsuttu. Tämän jälkeen komponenttiin syötetään generoitua tietoa ja painetaan nappuloita uudestaan. Tällaiset testit paljastivat hyvin, jos oli jonkun kentän tarkistuksessa virhe.

```
describe("should not crash when on table view", () => {
  it("should not crash when buttons are pressed", () => {
    const buttons1: [HTMLElement] = fixture.nativeElement.querySelectorAll("button");
    spyOn(component, "createNew");
    for (const b of buttons1) {
      b.click();
    }
    expect(component.createNew).toHaveBeenCalled();
    us.currentUser$.next(mockUser());
    component.currentConcept = mockConcept();
    const userList = [mockUser()/*, mockUser(), mockUser(), mockUser()*/];
    for (const u of userList) {
      u.profilePic = "";
    }
    component.dataSource = new MatTableDataSource(userList);
    fixture.detectChanges();
    const buttons2: [HTMLElement] = fixture.nativeElement.querySelectorAll("button");
    for (const b of buttons2) {
      b.click();
    }
  });
});
```

Kuva 22 Yksi käyttäjänäkymän Protractor-testeistä

5.3 Lopputulos

Lopputuloksena syntyi paljon testejä. Nämä testit tulevat toimimaan esimerkkinä ja pohjana tulevalle yksikkötestien ja testaamisen kehittämiselle. Uskon näiden testien olevan hyviä esimerkkejä miten tiettyjä osia kannattaa testata ja mitä osia ei kannata testata ollenkaan. Myös virheet vähenivät selvästi, mutta niin kuin usein testauksen kannattavuudessa, on vaikea määrittää, olivatko testit tämän takana. Itse uskoisin, että testien kehittämisen takia, yleisen ohjelmistokehityksen ymmärtämisen kasvu, on myös vaikuttanut tähän. Vain pidempi tarkasteluväli ja useampi projekti voi antaa todellisen varmuuden testien kehittämisen kannattavuudelle.

6 YHTEENVETO

Opinnäytetyön tavoitteena oli tutkia Angular-projektin testaamista ja kehittää valmiina olevaan projektiin käytettäviä testejä sekä luoda pohja näiden testien jatkokehittämiselle. Tarkoituksena oli myös, että tätä työtä voisi käyttää myös pohjana sekä tulevien, että nykyisten projektien testaamiselle. Sekä tavoitteena oli myös tietysti inhimillisten virheiden vähentäminen.

Tuloksena syntyi tietysti paljon testejä, mutta tärkein on mitä niillä saavutettiin. Inhimillisten virheiden määrä väheni selvästi. Angular ei ole helppo ohjelmistokehys aloittelevalle eikä sen testaaminenkaan ole sen helpompaa. Uskoin osaavani käyttää Angularia ennen tätä projektia, mutta projektin aikana huomasin, miten paljon paremmin tietyt asiat olisi voinut tehdä. Niin kuin moni muukin ohjelmistokehys, vaatii tämäkin paljon harjoitusta hallitakseen sen kokonaan. Uskon näiden testien kirjoittamisen auttaneen ymmärtämään paremmin, miten koko Angular toimii.

Kun alla oli jo valmista koodia, testien kirjoittaminen aiheutti paljon koodin uudelleen kirjoittamista. Tämän projektin pohjalta väittäisin, että jos aikoo kirjoittaa testejä mukaan projektiinsa, pitäisi se tehdä enne koodin kirjoittamista ylimääräisen työn välttämiseksi. Toisaalta kun omaa koodia käsittelee tällä tavalla jälkeenpäin, tulee oppineeksi virheistään eikä enää toista samoja virheitä.

Vaikka virheiden määrä väheni, silti niitä löytyi. Jos tekisin kaiken uudestaan, käyttäisin enemmän tarkkaavaisuutta ja huolellisuutta testien kirjoittamiseen ja etenkin niiden suunnitteluun. Testejä on paljon helpompi tehdä, kun on hyvä suunnitelma alla ja tehtyjen virheiden määrä on silloin myös alhainen. Rainmaker APP kehittyi myös tämän opinnäytetyön aikana, jolloin syntyi myös uutta koodia. Tätä uutta testattavaa koodia joutui kirjoittamaan uudestaan, mikä olisi voitu välttää, jos kirjoitetut testit olisi tehnyt ensin.

Jatkokehitettävää olisi testien laadun parantaminen ja parempi modularisointi. Sekä näiden testien jatkaminen myös Rainmaker APP:n muihin osiin, mutta en usko kehittäväni testejä kyseiseen projektiin ylimääräisen työn takia, koska en usko asiakkaan olevan valmis maksamaan siitä. Mutta uuden projektin koittaessa harkitsen testivetoista kehitystä.

Loppujen lopuksi koen näiden testien kirjoittamisen todella hyödylliseksi. Angular tarjoaa hyvät kirjastot ja työkalut näiden kehittämiseen ja jos vain on resursseja opetella, auttaa se ymmärtämään ohjelmistokehitystä, parantamaan koodin laatua ja vähentämään virheitä.

LÄHTEET

Angular 2018. One framework. Mobile & desktop. Viitattu 19.4.2018. https://angular.io/

AngularJS 2018. Guide to AngularJS Documentation. Viitattu 21.5.2018. https://docs.angularjs.org/guide

Andras, S. 2017. JavaScript Frameworks, why and when to use them. Viitattu 24.4.2018. https://blog.hellojs.org/javascript-frameworks-why-and-when-to-use-them-43af33d0608d

Ben 2017. JavaScript unit testing frameworks: Comparing Jasmine, Mocha, AVA, Tape and Jest. Viitattu 25.4.2018. https://raygun.com/blog/javascript-unit-testing-frameworks/

Dalke, A. 2013. Problems with TDD. Viitattu 19.4.2018. http://dalkescientific.com/writings/diary/archive/2009/12/29/problems_with_tdd.html

Graetz, B. 2018. A brief history of single-page applications. Viitattu 25.4.2018. http://angularjs-emberjs-compare.bguiz.com/mvc/spa-history.html

Haltiapuu, I. 2016. Testivetoinen Java-kehitys. Viitattu 19.4.2018.

Hussain, A. 2017. Angular Unit Testing. Viitattu 19.4.2018. https://codecraft.tv/courses/angular/unit-testing/jasmine-and-karma

Karma 2018. Karma – Spectacular Test Runner for JavaScript. Viitattu 25.4.2018. https://karma-runner.github.io/2.0/index.html

Kolodiy, S. Unit Tests, How to Write Testable Code and Why it Matters. Viitattu 26.4.2018. https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters

learnRxjs 2018. Viitattu 19.4.2018. https://www.learnrxjs.io/

Microsoft 2018. TypeScript – JavaScript that Scales. Viitattu 19.4.2018. https://www.typescriptlang.org/

Neuhaus, J. 2017. Viitattu 19.4.2018. https://medium.com/unicorn-supplies/angular-vs-react-vs-vue-a-2017-comparison-c5c52d620176

Protractor 2018. Protractor – end to end testing for Angular. Viitattu 19.4.2018. https://www.protractortest.org/#/

Reefnet, A. 2014. Answer to: “Is unit testing worth the effort”. Viitattu 19.4.2018. https://stackoverflow.com/questions/67299/is-unit-testing-worth-the-effort

React 2018. React – A JavaScript library for building user interfaces. Viitattu 24.4.2018. https://reactjs.org/

SWTF. 2018. Unit Testing. Viitattu 21.5.2018. http://softwaretestingfundamentals.com/unit-testing/

Trivedi, J. 2016. Basic Architecture of Angular 2 Applications. Viitattu 19.4.2018. https://www.c-sharpcorner.com/article/basic-architecture-of-angular-2-applications/